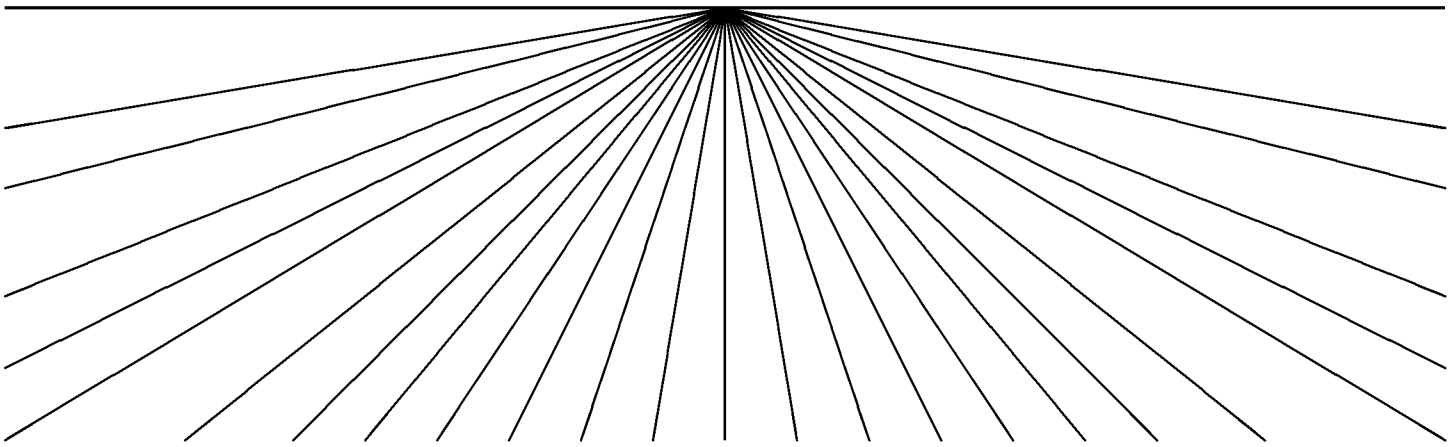
facultad de informática

universidad politécnica de madrid

Analysis of Concurrent Constraint Logic Programs with a Fixed Scheduling Rule

Francisco Bueno
Manuel Hermenegildo



Analysis of Concurrent Constraint Logic Programs with a Fixed Scheduling Rule

Keywords

Concurrent Constraint Programming, Program Analysis, Abstract Interpretation, Andorra Model

Abstract

The analysis of concurrent constraint programs is a challenge due to the inherently concurrent behaviour of its computational model. However, most implementations of the concurrent paradigm can be viewed as a computation with a fixed scheduling rule which suspends some goals so that their execution is postponed until some condition awakens them. For a certain kind of properties, an analysis defined in these terms is correct. Furthermore, it is much more tractable, and in addition can make use of existing analysis technology for the underlying fixed computation rule. We show how this can be done when the starting point is a framework for the analysis of sequential programs. The resulting analysis, which incorporates suspensions, is adequate for concurrent models where concurrency is localized, e.g. the Andorra model. We refine the analysis for this particular case. Another model in which concurrency is preferably encapsulated, and thus suspensions are local to parts of the computation, is that of CIAO. Nonetheless, the analysis scheme can be generalized to models with global concurrency. We also sketch how this could be done, and we show how the resulting analysis framework could be used for analyzing typical properties, such as suspension freeness.

Contents

1 Introduction

We discuss in a step–wise manner an analysis algorithm based on abstract interpretation which can be used to analyze concurrent execution of (constraint) logic programs. Our approach is based on considering a fixed computation rule (e.g., left-to-right for sequential execution) with possible suspension of goals during the computation. Several approaches in these style have been proposed in [DGB94] and [MdlBH94] (see also [dlBMS95]). We use as a starting point that of [DGB94] and adapt it to the special case of the Andorra concurrent model. The scheme is also applicable to generic concurrent models, as well, and can derive in analyses of properties such as suspension freeness [CFM91, CMFW93].

Motivated by the desire to support concurrent execution in sequential machines, and still apply many optimizations which can be applied to languages with a purely sequential computation rule (as in [RD90]), we explore the application of this kind of analyses in this task. In the case of the Andorra model, its main application is in detecting determinacy of goals at compile–time. In this model, determinate goals are executed first and concurrently, non–determinate goals being delayed until a second phase [SCWY90]. In this phase, the leftmost suspended goal is awakened, and a new round of computation is started. The analysis can be refined to take this model into account. In some cases, a compile–time scheduling [KS92] of determinate goals can be performed.

The advantage of considering a left-to-right computation rule is that classical analysis technology for sequential programs can still be applied if suitably extended to take care of possible suspension. We sketch how this can be done in the case of the Andorra model and for the general case of concurrent programming. The discussion is implementation oriented since our intention is to provide an easy implementation path by using existing technology.

2 Weak Non–suspension Analysis

We use as starting point the *Weak Non–suspension Analysis* defined in [DGB94], and propose some improvements. This analysis is aimed at inferring literals in a program whose complete execution will not suspend. To do this, flags $p_i.nosusp$ are attached to every literal p_i and a flag $p.nosusp$ to every predicate p. The conditions on a goal to not suspend are given in terms of the *demand* of the corresponding predicate. The demand of a predicate is a sufficient condition on the variables of its arguments which guarantees that a reduction is taken. It is defined in the abstract domain of analysis *ASub* by:

$$demand(\mathtt{p}) = \sqcap \{\lambda \in ASub \mid \forall c \in \gamma(\lambda)\ \nexists t \in \rightarrow \text{ s.t. } t \text{ is a suspension transition for } \mathtt{p}\}$$

A reduction of a goal of p is taken in an abstract environment λ if $demand(\mathtt{p}) \sqsubseteq \lambda$.

The analysis procedure is based on an underlying (top–down) left-to-right analysis of the program. This analysis is given by a function $analyse\_call(p(\bar{u}), \alpha)$ which performs the analysis of a call pattern $\langle p(\bar{u}), \alpha \rangle$ and gives the resulting success substitution. To facilitate things, we parameterize the presentation of the algorithm with the following domain dependent functions:[1]

- $extend\_abs\_env(\alpha, \alpha')$ which yields the extension of an abstract substitution α' accordingly to another abstract substitution α which describes a particular (abstract) environment,
- $call\_to\_entry(p(\bar{u}), p(\bar{v}), \alpha)$ which gives an abstract environment describing the effects on $\bar{u}$ of unifying $p(\bar{u})$ with $p(\bar{v})$ given an abstract substitution α describing $\bar{v}$,
- $exit\_to\_success(\alpha', p(\bar{u}), C, \alpha)$ which gives an abstract environment describing $\bar{u}$ w.r.t. α (which describes $vars(C)$) and the effects of unifying $p(\bar{u})$ with the head of C under the abstract environment α' describing $\bar{u}$,
- $project(\bar{u}, \alpha)$ which projects the abstract substitution α over the variables $\bar{u}$.

Let the abstract environment at the program point immediately before the literal $L_i \equiv q_i(\bar{u}_i)$ be denoted by A_i, $1 \leq i \leq n$. The analysis proceeds via a fixpoint iteration of a function computing the collecting semantics for each clause of the program, i.e. the abstract environments attached to each program point. The analysis algorithm for each clause $C \equiv p(\bar{u})$:- $L_1, \ldots, L_n$ of the program is given in Figure 1.

$analyse\_clause(p(\bar{v}), C, A) \equiv$
  $A_1 := call\_to\_entry(p(\bar{u}), p(\bar{v}), A)$;
  For $i := 1$ to n do
    $A'_i := analyse\_call(q_i(\bar{u}_i), A_i)$;
    $A''_i := extend\_abs\_env(A_i, A'_i)$;
    $L_i.nosusp := q_i.nosusp \wedge (A_i \sqsupseteq demand(q_i))$;
    $A_{i+1} :=$ **if** $\neg L_i.nosusp$ **then** $A_i \sqcup A''_i$ **else** A''_i;
  od;
  $p.nosusp := p.nosusp \wedge (\wedge_{i=1}^{n} L_i.nosusp)$;
  return $project(\bar{u}, A_{n+1})$;

Figure 1: Weak Non–Suspension Analysis

This algorithm gives $L_i.nosusp = false$ whenever the corresponding predicate flag $q_i.nosusp = false$. This may cause loss of precision, since some call sites of q_i may

[1] These functions are defined in terms of the abstract domain supporting the analysis.

be suspending and some not, but as long as one of them is, then $q_i.nosusp = false$ and because of this, in successive iterations of the algorithm, all call sites will end up as $false$. To remedy this, we can use a proper flag to carry the *nosusp* bit of the descendants up to the parent, and avoid the use of the predicate flag for this purpose. An algorithm in this style is given in Figure 2. To obtain the desired effect, the algorithm is in fact a projection of that of Figure 1 onto the underlying analysis, instead of blindly relying on it. The algorithm is thus given in terms of two mutually recursive functions, one for clauses and one for literals.

$analyse\_clause(p(\bar{v}), C, A) \equiv$
  $A_1 := call\_to\_entry(p(\bar{u}), p(\bar{v}), A)$;
  For $i := 1$ to n do
    $(A'_i, L_i.nosusp) := analyse\_literal(q_i(\bar{u}_i), A_i)$;
    $A''_i := extend\_abs\_env(A_i, A'_i)$;
    $L_i.nosusp := L_i.nosusp \wedge (A_i \sqsupseteq demand(q_i))$;
    $A_{i+1} :=$ **if** $\neg(A_i \sqsupseteq demand(q_i))$ **then** $A_i \sqcup A''_i$ **else** A''_i; (1)
  od;
  return $(project(\bar{u}, A_{n+1}), \wedge_{i=1}^{n} L_i.nosusp)$;

$analyse\_literal(q(\bar{u}), A) \equiv$
  $A_L := project(q(\bar{u}), A)$;
  $A'_L := \bot$;
  $L.nosusp := true$;
  For each clause C which matches $q(\bar{u})$ do
    $(A_C, C.nosusp) := analyse\_clause(q(\bar{u}), C, A_L)$;
    $A'_L := A'_L \sqcup exit\_to\_success(A_L, q(\bar{u}), C, A_C)$;
    $L.nosusp := L_i.nosusp \wedge C.nosusp$;
  od;
  return $(A'_L, L.nosusp)$;

Figure 2: Weak Non–Suspension Analysis: flags moved into descendants

Additionally, the algorithm in Figure 2 yields the lub of the abstract environments A_i and A''_i only when $L_i.nosusp = false$ is caused by the possible suspension of L_i itself, not if it is caused by some descendant of it. This can be done because the effects of the possible suspension of the descendant will be taken into account when analyzing this descendant, and properly propagated upwards. Furthermore, doing this can potentially increase precision. Let $L_i \equiv p(x, y)$, and the descendant be $q(x)$, the lub at the point where $q(x)$ is analyzed will probably cause x to "go to top", but not y, which will be caused if doing the lub at the point where $p(x, y)$ is analyzed.

Note that the rationale behind such a lub operation is that of taking into account

the effects of the (possible) suspension of the literal as well as those of its (possible) execution. This has to be done due to the fact that the non–suspension flag can not be safely determined to be *true*. However, for the only purpose of non–suspension it is equivalent to leave these variables in the less instantiated state, i.e. that of "before the execution," which is that of the calling abstract environment. The only modification required to the algorithm in Figure 2 is to replace (1) by:

$$A_{i+1} := \textbf{if } \neg(A_i \sqsupseteq \mathit{demand}(q_i)) \textbf{ then } A_i \textbf{ else } A_i'';$$

Whereas the original algorithm took the worst case possible by doing the lub of the call substitution and the underlying success substitution when there was a possibility of the literal being suspended, the new algorithm simply does not consider the underlying substitution. This is safe as long as downwards closed properties (in the lattice of substitutions) are analyzed. In what regards non–suspension, demands are usually defined in terms of state of instantiation, and the effects of the lub w.r.t. this is to lose any information about the state of the affected variables. However, using the call substitution instead has the same effect, and can make the analysis more efficient.

3 Determinacy Analysis

When supporting the Andorra style coroutining in the computation model (as in Andorra-I [SCWY90] or AKL [JH91]) a key issue is determinacy. The Andorra model runs determinate goals concurrently. For this purpose determinacy has to be checked at execution time. Thus, analysis can help in determining goals which are known to be determinate, reducing run–time overheads.

We observe that Weak Non–suspension can be used for these purposes just by defining demand in terms of the determinacy condition of the predicates. Furthermore, we can redefine the *nosusp* flags, because all that we are interested in is the demandness check: if a reduction of a goal can be done deterministically. The flag can be set to the result of this check, and this can be either done during the analysis or simply reconstructed after the analysis has finished. There is no need of carrying non–suspension conditions throughout all the analysis. Such an analysis will correspond to the execution of the *first* determinate phase of the Andorra execution model. Woken up goals can be ignored because they do not change the state of instantiation of variables in such a sense that something which could run (i.e., did not suspend) will now, because of the awakening of this goal, suspend. This is guaranteed by the monotonicity of the computation.

If we consider the predicate level then we can use the algorithm of Figure 2. Additionally, if we assume the above considerations (i.e., that for the purpose of non–suspension doing the lub is equivalent to ignoring the exit substitutions), we end up with the algorithm of Figure 3.

Note that now analyzing a literal which cannot be guaranteed to proceed may not

$analyse\_clause(p(\bar{v}), C, A) \equiv$
  $A_1 := call\_to\_entry(p(\bar{u}), p(\bar{v}), A);$
  For $i := 1$ to n do
    $L_i.nosusp := (A_i \sqsupseteq demand(q_i));$
    $A_{i+1} :=$ **if** $L_i.nosusp$ **then**
      $analyse\_literal(q_i(\bar{u}_i), A_i)$ **else** A_i;
  od;
  return $project(\bar{u}, A_{n+1})$;

$analyse\_literal(q(\bar{u}), A) \equiv$
  $A_L := project(q(\bar{u}), A);$
  $A'_L := \bot;$
  For each clause C which matches $q(\bar{u})$ do
    $A_C := analyse\_clause(q(\bar{u}), C, A_L);$
    $A'_L := A'_L \sqcup exit\_to\_success(A_L, q(\bar{u}), C, A_C);$
  od;
  return $extend\_abs\_env(A, A'_L)$;

Figure 3: Determinacy Analysis at the predicate level

be required. In Figure 3 these literals are not analyzed. However, care has to be taken of the non–suspension flags, assuming them as $false$ in the whole subtree under these literals. Alternatively, the algorithm can be adapted to analyze all clauses, whether they are non–suspending or not, but with a substitution $\top$ if they are not. The effect of this is to take into account the propagation of guards if/when the clauses which the analysis can not determine to be non–suspending do not suspend in fact. Such an analysis could be used (and possibly be worth doing) even in the absence of query modes or entry points (while the top–down scheme sketched will probably yield unuseful information in such a case). The algorithm of Figure 3 can be completed by doing:

For each literal $q_i(\bar{u}_i)$ in the program s.t. $q_i.nosusp = false$ do
  $analyse\_literal(q_i(\bar{u}_i), \top);$

This is safe in any scheduling policy for concurrency. But for a default left-to-right rule we can do better. Every non–suspending goal to the left of a literal $\mathtt{q_i}$ will be executed before $\mathtt{q_i}$. Therefore, again for downwards closed properties, assuming the abstract environment computed at the point of calling $\mathtt{q_i}$ is always safe. The modification to the algorithm appears in Figure 4.

By observing that determinacy conditions are usually a disjunction of conditions

$analyse\_clause(p(\bar{v}), C, A) \equiv$
  $A_1 := call\_to\_entry(p(\bar{u}), p(\bar{v}), A)$;
  For $i := 1$ to n do
    $A'_i := analyse\_literal(q_i(\bar{u}_i), A_i)$;
    $L_i.nosusp := (A_i \sqsupseteq demand(q_i))$;
    $A_{i+1} :=$ **if** $L_i.nosusp$ **then** A'_i **else** A_i;
  od;
  return A_{n+1};

Figure 4: Determinacy Analysis with guard propagation

which are exclusive of each other (since they are based on mutual exclusion of clauses) we can also think of applying the analysis for separate clauses. There are alternative ways to detect determinacy. For example, in flat determinacy only head unification and simple tests are taken into account. Consider the well-known Fibonacci program,

```
fib(0,1).
fib(1,1).
fib(A,B) :- A>1, ...
```

In this program a goal will be determinate if the first argument is bound or the second one is different of 1. This can be expressed as a demand of the `fib(A,B)` predicate which approximates `(nonvar(A) ; nonvar(B))`. In a language with left-to-right computation rule and suspension primitives (see [BDdlBH95]) we can write this as:

```
fib(A,B,S,L,L1):- nonvar(S), !, fib_det(A,B,L,L1).
fib(A,B,S,L,L1):- nonvar(A), !, fib_det(A,B,L,L1).
fib(A,B,S,L,L1):- nonvar(B), !, B\==1 -> fib_det(A,B,L,L1) ; ...
...
```

It is clear that if the demand of the second clause is known to hold there is no need to consider the others.

Each clause has an attached determinacy condition, and we can use this to improve the analysis. Since conditions are exclusive, we can expect some improvement from keeping the analysis of each clause separate. When a clause is found not to suspend we "commit" to it during the analysis. The rest of the clauses can be ignored. Such an algorithm will look like that of Figure 5.

We can now define the demand of each clause as just the abstraction of its guard (considering the above tests as guards), and use the algorithm in Figure 5 with this concept. Whenever one of these clauses is non–suspending, we analyse this one and only

$analyse\_clause(p(\bar{v}), C, A) \equiv$
  $A_1 := call\_to\_entry(p(\bar{u}), p(\bar{v}), A)$;
  For $i := 1$ to n do
    $(A'_i, L_i.nosusp) := analyse\_literal(q_i(\bar{u}_i), A_i)$;
    $A_{i+1} := A'_i$;
  od;
  return A_{n+1};

$analyse\_literal(q(\bar{u}), A) \equiv$
  return **if** $\exists C \mid (A \sqsupseteq demand(C))$
    then $(analyse\_clause(q(\bar{u}), C, A), true)$
    else $(A, false)$;

Figure 5: Determinacy Analysis with clause selection

this one clause, since they are exclusive. There is no need to analyze non–intervening clauses. In other cases, lub can always be replaced by carrying variants to improve precision by specialization.

4 Extension to Sequential Concurrent Programs

In the previous algorithms a conservative estimation of the computation environments which can occur at each point at execution time was inferred. Nonetheless, this estimation is too conservative. Better estimations can be obtained by re–analyzing those literals which have not been found to not suspend. The rationale for this is that, once that one left-to-right pass of the computation has been performed, several goals may be woken. The algorithms proposed do not take this into account.

In [DGB94] a refinement of weak non–suspension analysis is given, called *strong* non–suspension. Strong non–suspension guarantees that goals which might be suspended are not awoken during the computation of weakly non–suspending goals. This makes the analysis more tractable. However, it is too restrictive. On the contrary, in [MdlBH94] possibly suspended goals are tracked during the analysis, which requires and additional level of abstraction. This can improve precision of the analysis, but can also add too much overhead to it. We follow the approach of [CCC90], in which possibly awoken goals are taken care of by "re–circulating" the abstract environment resulting from the first pass of analysis.

As opposed to the analysis of [CCC90], we don't re–analyze each clause in turn starting with the abstract environment resulting from the analysis of that same clause.

Instead, we complete a fixpoint of the first left-to-right pass over the whole program, and then restart the analysis with the resulting abstract environment. Because of this, we need only analyze again those literals which were not found to not suspend during the previous analysis round. An algorithm which does this is given in Figure 6.

$$
\begin{array}{l}
\mathit{analyse\_clause}(p(\bar{v}), C, A) \equiv \\
\quad A_1 := \mathit{call\_to\_entry}(p(\bar{u}), p(\bar{v}), A); \\
\quad \text{For } i := 1 \text{ to } n \text{ do} \\
\qquad A'_i := \textbf{if } L_i.\mathit{nosusp} \textbf{ then } A_i \textbf{ else} \\
\qquad\qquad \mathit{analyse\_literal}(q_i(\bar{u}_i), A_i); \\
\qquad L_i.\mathit{nosusp} := \textbf{if } L_i.\mathit{nosusp} \textbf{ then } \mathit{true} \textbf{ else} \\
\qquad\qquad (A_i \sqsupseteq \mathit{demand}(q_i)); \\
\qquad A_{i+1} := \textbf{if } L_i.\mathit{nosusp} \textbf{ then } A'_i \textbf{ else } A_i; \\
\quad \text{od}; \\
\quad \text{return } A_{n+1};
\end{array}
$$

Figure 6: Sequential Concurrent Analysis

Note that clauses which may suspend are analyzed altogether, as in the algorithm of Figure 4. Clauses which are found not to suspend in successive rounds of analysis will then be analyzed with new abstract environments. The resulting computed information is safe in the sense of approximating all environments which can occur at run–time independently of whether the clause suspends or not (again, for downwards closed properties). However, the *nosusp* flags are changed from one round to another, in order for the analysis to decrease and allow a fixpoint to be eventually reached. The resulting flags after such fixpoint is reached cannot be relied upon in the sense of the left-to-right computation. Nonetheless, whenever all flags are set to *true* at the end of the analysis, suspension freeness [CFM91] of the program is guaranteed. Consider the following program modelling a producer/consumer:

```
p(X):- | X=[a|Y], p(Y).
p(X):- | X=[].
```

```
q(X):- X=[a|Y] | q(Y).
q(X):- X=[] |.
```

The analysis of a query "`q(X), p(X).`" will proceed as follows. `X` is free upon entering `q(X)` and thus $q.\mathit{nosusp} = \mathit{false}$, and `X` is assumed free upon entering `p(X)`. Then $p.\mathit{nosusp} = \mathit{true}$, and `X` is found to be ground upon success of `p(X)`. A fixpoint is reached with this information. Since $p.\mathit{nosusp} = \mathit{true}$, `p(X)` is found to be executed in a left-to-right fashion, allowing possible optimizations. A second round of analysis will start with `X` being ground. Then $q.\mathit{nosusp} = \mathit{true}$, and `X` continues being ground upon success of `q(X)`. `p(X)` is not analyzed. An overall fixpoint is reached at this point. Since all flags are found to be *true*, the program is guaranteed to be suspension free.

5 Conclusions

We have given succesive refinements of an analysis algorithm originally defined for sequential (left-to-right) programs which can be applied in different fashions to the analysis of concurrent models. Weak non–suspension analysis can be used to identify computation threads which do not suspend, and therefore can be executed in a sequential fashion (as in scheduling analysis [KS92]). This allows performing many optimizations which are possible in the sequential computation (as demostrated in [DGB94]. Our refinement of this analysis could further improve the number of cases in which this is possible.

A second refinement is possible when considering the Andorra model. In this case, the analysis is useful for detecting determinacy at compile–time, allowing to avoid the run–time detection overhead. Similar applicability could be found in computation models such as CIAO, where concurrency can be encapsulated to parts of the computation. Finally, the scheme can also be applied to standard concurrent constraint models.

All algorithm definitions given rely on the core functions of classical analysis frameworks for sequential languages, e.g. PLAI [MH90, BdlBH94]. Therefore, the implementation effort can be reduced by just modifying PLAI so that it takes into account the management of the non–suspension flags.

References

[BDdlBH95] F. Bueno, S. Debray, M. García de la Banda, and M. Hermenegildo. Transformation-based Implementation and Optimization of Programs Exploiting the Basic Andorra Model. Technical Report CLIP11/95.0, ACCLAIM Deliverable D3.3/4-A1, Facultad de Informática, UPM, May 1995.

[BdlBH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical Report CLIP2/94.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, February 1994.

[CCC90] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *North American Conference on Logic Programming*, pages 215–232, October 1990.

[CFM91] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis of Concurrent Logic Programs. In *Eighth International Conference on Logic Programming*, pages 331–345, Paris, France, June 1991. MIT Press.

[CMFW93] M. Codish, K. Marriott M. Falaschi, and W. Winsborough. Efficient analysis of concurrent constraint logic programs. In *Twentieth International Coll. Automata, Languages and Programming*, Lund, Sweden, July 1993.

[DGB94] S. Debray, D. Gudeman, and Peter Bigot. Detection and Optimization of Suspension-free Logic Programs. In *1994 International Symposium on Logic Programming*, pages 487–501. MIT Press, November 1994.

[dlBMS95] María García de la Banda, Kim Marriott, and Peter Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. Technical Report CLIP9/95.0, ACCLAIM Deliverable D3.2/3-A1, Facultad de Informática, UPM, March 1995.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[KS92] Andy King and Paul Soper. Serialisation analysis of concurrent logic programs. In Giorgio Levi and Hélène Kirchner, editors, *Algebraic and Logic Programming, Third International Conference*, LNCS 632, pages 322–334, Volterra, Italy, September 2–4, 1992. Springer-Verlag.

[MdlBH94] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[MH90] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[RD90] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, April 1990.